Critical Reference Counting

P.A. Jones, C.H.A. Koster, P. van Bommel, Th.P. van der Weide

Computing Science Institute/

CSI-R9825 September 1998

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Critical Reference Counting

P.A. Jones, C.H.A. Koster, P. van Bommel, and Th.P. van der Weide

Computing Science Institute, Katholieke Universiteit Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, paulj@cs.kun.nl *

Abstract. In traditional languages such as Pascal, C and C++ the management of allocation and deallocation of memory is left to the programmer. Leaving such a task to the programmer is error prone and cumbersome. In many recent programming languages the compiler performs this task for the programmer. However compiler-writers often implement automatic memory management as an add-on, going out of their way to minimize the effect on the normal logic of the compiler. Many authors have found that automatic memory management can be improved by compile-time analysis. Automatic memory management may be treated as yet another optimization, trading memory for speed, much like the traditional optimizations that can be done for stack-based architectures such as lifetime analysis, data flow analysis and sharing of identical values. We introduce the technique of Critical Reference Counting in combination with compile time analysis in order to reduce the overhead of memory management. A comparitive study shows that our system only slightly increase runtimes of programs, whereas more conventional methods increase program runtimes substantially.
Keywords. automatic memory management, reference counting.

* Currently employed at Edmond Research & Development b.v. Toernooiveld 220, 6525 EC Nijmegen, The Netherlands

1 Introduction

The management of allocation and deallocation of memory is traditionally left to the programmar. The past experience of programming in traditional languages such as Pascal,C and C++ have shown that programmars often make mistakes in memory management. For this reason in many recent programming languages the compiler performs this task for the programmer. Part of the task of an automatic memory manager is to keep track of which parts of memory are still needed and which part of memory may be reused. This task is called `garbage collection`.

There exist several methods to do automatic memory management, but they all fall into two categories: the *continuous* and *discontinuous* memory management systems. A discontinuous system typically has an detection phase in which it examines all memory to determine which parts may be reused. The detection phase is usually started when an allocation request would fail. In a continuous system the administration of which parts of memory might be used is updated at each assignment, therefore immediatly detecting parts of memory no longer needed . For an overview of the types of automatic memory management or garbage collectors see [Knu69,App91,Wil92], further details on discontinuous garbage collectors can be found in [McC60,FY60,Che70,App89,App91,LH83].

Both continuous and discontinuous memory management systems have their advantages and disadvantages. A continuous system uses the smallest possible amount of memory, but usually incurs a high cost due to the overhead at each assignment. A discontinuous system has a low overhead during normal program execution, but uses more memory than strictly necessary and the overhead is clustered in the detection phase which may take quite some time. Currently the discontinuous systems are more widespread in use due to several facts:

1. Most dicontinuous systems need little or no support from the compiler itself.
2. Given more memory the overhead of the memory management system becomes smaller. This is mainly due to the fact that the detection phase occurs less frequent.
3. Discontinuous systems can handle arbritrary types of datastructures, including circulair ones. Not all continuous systems can deal with circular datastructures.

The memory management system proposed in this paper is intented to be used for functional languages (e.g. MIRANDA), logical languages (e.g. PROLOG) and attribute/affix grammar based languages (e.g. AGFL) excluding circulair datastructures. The rest of the paper will show how common properties of these languages can be used to construct a continuous memory management system that outperforms other systems.

The choice for a continuous memory management system, was made for the following reasons:

1. minimal memory requirements, which is essential for smaller or more heavily loaded larger machines where excess memory usage can have a serious impact on performance due to virtual memory overhead.

2. distributed overhead, which is especially important for interactive and real-time programs. In such programs halting the program for a longer period in order to do memory management is not acceptable.

First a more detailed language model will de given for which an automatic continuous memory management system will be derived. Then a brief introduction into Reference Counting methods follows. This is followed by a description of how a Reference Counting system can be made more efficient by using life-time information available to a compiler. The resulting *Critical Reference Counting* method is then shortly compared to Deferred Reference Counting methods. A more detailed comparison between the different memory management systems is given based on experimental results showing how well the Critical Reference Counting method performs. This paper ends with some onclusions and ideas for future research.

2 Language Model

We have to make some assumptions about the language model in order to be able to use properties of this language to optimize memory management. In this paper we limit ourselves to strongly typed languages with a call by value/result parameter passing semantics. Furthermore it is assumed that the language does not provide reference semantics. This language model unfortunatly excludes JAVA, but it does include a broad family of functional and applicative languages, and also ADA.

Due to the exclusion of reference semantics in the language model, circular data structures, aliases and partial value updates need not be considered. Note that although reference semantics are excluded the compiler *can* use (and most probably will use) references in the compiled code.

To be able to reason about programs written in such a language a general type system is introduced with commonly used constructs. Most type systems provide the following constructs:

1. *primitive* types, i.e. `INTEGER`, `BOOLEAN`, `TEXT` and *simple* values, considered a type with one instance (e.g. nil)
2. *compound* types, combining a tuple of values into one value (Cartesian product)
3. *union* types, allowing a choice of types (Cartesian sum)

User defined types are specified in terms of these basic constructs and other user-defined types. In general recursive types may be allowed, so that lists and tree-like structures can be described. Here a syntactic notation will be used to specify types. A user-defined type is described by context-free rules with type-names as nonterminals and simple types as terminals. Throughout this paper nonterminals are written in uppercase and terminals in lowercase. A compound type is denoted by writing its component types in sequence. A union type is denoted by its alternative types, separated by semicolons, where the order plays no role. Using

this notation a list of integers can be specified as: `LIST :: INT LIST; nil.` which should be read as: a list is either an integer followed by another list or it is the constant nil.

The representation of values in memory could be achieved using several methods. A simple minded approach would be to represent a value in a continuous piece of memory. However it would then be impossible to determine at compile time how much memory would be needed for a value of a recursive type such as the aforementioned `LIST` type. This problem can be solved by implementing a variable as a reference to a value and allocating the value somewhere in the *heap*. In this simple approach frequently used operations such as copying are expensive, requiring data to be moved in memory.

Avoiding copying of data, and thus hopefully speeding up the copying operation, can be achieved by copying not the value itself but only the reference to it. Also during the creation of a value a large amount of data might have to be copied, especially in the construction of *compound* values. Therefore, applying the same reasoning as before, each part of a compound value should be a reference to the actual value rather than a copy. Further optimizations include preallocating certain constant values, so that they have constant references. This fact can then be exploited in equality tests. However this *sharing* of values introduces the problem of knowing when the memory occupied by a value is no longer needed.

In our model, a value is implemented as a piece of memory, possibly containing references to other values. In order to be able to do memory management, some definition of whether a value is needed or not must be established. Usually a conservative approach is used where a value is deemed to be *accessible* (or needed) if a reference to it can be obtained by some combination of operations starting with a reference stored in directly accessible memory such as the stack or the registers of the processor. This is a conservative approach, in the sense that a value may be deemed accessible while in fact the program might not actually access it. Even though this approach is an approximation, it is *safe* (a value is never deemed inaccesible when it in fact isn't) and fairly *accurate* (not too many values are deemed accesible when they in fact are not).

3 Reference Counting

Most continuous memory management systems employ a form of *reference counting* [Col60]. The main idea is to keep for each value a count of the number of references to that value. As long as this *reference count* is larger than zero, the value is deemed to be accessible . As soon as the reference count for a value reaches zero the value can no longer be accessible, as no reference to it exists. Now the memory occupied by that value can be reclaimed.

It is important to realize that the classical reference count algorithm can not reclaim circular data structures. This inability is due to the fact that every value in such a datastructure has at least one reference to it. Dealing with circular data

structures requires additional work, see [Kop85,Axf90]. Since our language model does not have reference semantics circular datastructures need not be considered.

Continuous memory management systems require a certain amount of work each time a reference is modified. Whenever a reference is created or copied, an *attach* operation must be performed, incrementing the reference count of the value referenced. Every time a reference is destroyed, a *detach* operation must be performed. The detach operation is more complex, as it comprises not only a decrease of the reference count but also a check whether this was the last reference. When the reference count reaches zero, the memory occupied by the value referenced should be reclaimed. Reclaiming a value means that any references stored in that value is destroyed. This must be taken into account, and therefore a detach operation must recursivly be performed for each of those embedded references. Note that this implies that the detach operation is type-dependent, as it must be known which parts of the value are a reference.

Due to the complexity and the recursive nature of the detach operation, it is common to create specialized detach functions for each type. Usually the attach operation is not implemented by a separate function. This is due to the generic nature of the attach operation and the small number of instructions needed to implement it.

It is clear that calling a function whenever a reference is destroyed creates an overhead. Reducing the overhead in the classical reference count algorithm is not possible, since the algorithm does not allow a reference to a value to exist without updating the reference count for that value upon every application. In fact the rules regarding the reference count of a value are too strict. The algorithm really only uses the fact that the reference count of a value is nonzero to decide whether the memory occupied by that value should be reclaimed. We will use this fact to derive the more efficient Critical Reference Counting method.

4 Critical Reference Counting

Relaxing the requirements on the value of a reference count in the classical reference count algorithm yields a new algorithm which is called *Critical Reference Counting*. In this algorithm the reference count of a value must be nonzero (rather than precisely the number of references) as long as there exists a reference to it. This relaxed requirement can be exploited if it is possible to determine statically that the reference count of a value will be nonzero in certain parts of a program. In those part of the program extra references to that value may exist without updating the reference count.

The question is whether it is possible to know statically that the reference count of a certain value is nonzero? By definition this can only be the case if at least one reference to that value exists. Therefore as long as such a reference exist copies of that reference can be used without modifying the reference count of the referenced value. In particular, copying a reference stored in a value does not need to update reference counts as long as a reference to that value exists.

Note that if it is impossible to determine statically whether such an extra reference exists, the reference count will have to be updated much like the Classical Reference Count algorithm. One might call such references *critical* as failing to update the reference count for them might result in premature reclamation of memory, hence the name *Critical Reference Counting*.

4.1 Detecting Critical References

Determining whether a particular reference is critical or not can be done using several methods. For instance: the semantics of parameter passing can be exploited if the caller is responsible for creating and destroying the copies that must be made while passing parameters. In that case it is known that during a call the caller will have references to each of the passed values. Therefore, during the call copies of references to the passed values may exist without updating the appropriate reference counts.

Application of the above can lead to a substantial reduction in the number of attaches and detaches that must be performed. For instance, a function that checks whether an element `i` is in a list `x` can do so without modifying a reference count. A recursive function `in` is specified in pseudo code as follows:

```
BOOL function in(x:LIST,i:INT)
var e: INT; t: LIST;
begin
   if [x->e t]
   then begin
            if i=e
            then return True
            else return in(t,i)
        end
   else return False
end
```

where the *guard* `[x->e t]` is a conditional assignment, first checking whether `x` is indeed of the form `INT LIST`. If this is the case, the variables are assigned the corresponding values (`e` becomes the head and `t` the tail of the list `x`).

Using the assumption that the caller keeps a reference to the values passed, it is known that upon calling the function `in` the values of `x` and `i` have a nonzero reference count. The values that might be assigned to `e` and `t` must also have a nonzero reference count due to the fact that the references in the value of `x` will exist until the end of the call. For the recursive call to `in`, copies can be made of the value of `t` and `i` without updating the reference counts. This is possible because the recursive call will complete before this call completes. Therefore the values in `t` and `e` are guaranteed to have a nonzero reference count until the call completes.

4.2 Result values

In the previous paragraphs only parameters and local variables were considered. Values that are returned from a call (*result values*) must also be dealt with by the memory management system. Unfortunately, the returning of values can not be optimized like parameter passing. The main cause is that if a reference to a value is returned there is no obvious candidate that is guaranteed to keep a copy of that reference for any period of time. Therefore the calling function must assume that it might just have received the last reference, and will eventually have to do a detach for that reference. This in turn implies that if the value returned was not a newly created value, the function called will have to do an attach for the reference to that value.

Usually there are, however, some functions which will always return a reference to a part of a value passed to it. Examples of such functions can be found in code that retrieves values from data structures such as symbol tables, lists and trees. In these case such a function returns a reference to a value contained in a complex datastructure. In those parts of the program where it is known that the datastructure is still needed (i.e. has a reference count of at least one) it is also known that at least one reference to the returned value exists (somewhere in the datastructure). This in turn implies that in those same parts of the program no action needs to be taken to adjust the reference count of the returned value.

For this purpose a relation must be established between the values passed and returned. This relation can provide the calling function with the extra knowledge needed to decide for which of the returned references a detach will be necessary.

4.3 Global variables

In one of the previous paragraphs, the assumption was made that the caller would be responsible for creating and destroying copies of values that are passed as a parameter. What must be done when the value of a global variable will be passed as a parameter? In fact this depends on whether the called function might also directly modify the global variable thorugh a *side effect*. If it does not modify that global variable, then it is known that during the call the copied value of the global variable will have at least one reference to it (the one in the global variable). As a consequence no attach and detach operations will be needed.

What must be done if the called function *can* modify the global variable? Modification of the global variable might execute a detach operation which might trigger the reclamation of the memory occupied by the referenced value. The memory occupied by the value should not be reclaimed before the call has ended. Therefore this copying must introduce an attach operation before the call and a detach operation after the call to prevent the premature reclamation of memory to occur. After the call a copy of the reference is needed to perform the detach operation. Therefore the function must store that reference in a local variable.

Knowing whether a function can or can not modify a particular global variable requires a global analysis. If a global analysis can not be performed or is

deemed too expensive, a *safe approximation* can be made using the module hierarchy. The approximation is based on the observation that a global variable can not be modified during a call to a function in a different module if that module does not directly or indirectly use the module in which the global variable is defined.

4.4 Tail-recursion removal

Tail-recursion occurs when the last operation in a function happens to be a call to the function itself and the result of the function is the result of the recursive call. The search algorithm presented earlier in this paper is such a tail-recursive function. It is a well known fact that such tail-recursive functions can be transformed into loops.

Unfortunately, the memory management system sometimes turns a tail-recursive function into a non tail-recursive function by introducing detach operations after the call. This might occur if the tail-recursive call is passed a newly created value or the value of a global variable. The detach operations are introduced because the calling function is responsible for creating and destroying copies of the values it passes to other functions.

It would be desirable to merge these detach operations into the recursive call. Fortunately, the recursive call has access to copies of the references on which these detach operations must be performed. Therefore it could be arranged that, before the return, the necessary detaches on the copied references are performed. Note that it is not possible to just add this code to the original tail-recursive function, as this would perform spurious detach operations on the original arguments.

Take as example:

```
LIST function intersect(x:LIST,y:LIST)
var t:LIST; h:INT;
begin
   if [x->h t]
   then return intersect(t,remove(h,y))
   else return y
end
```

It is clear that this function is tail-recursive. In the recursive call the second argument is a newly created list, the result of `remove(h,y)`. Making the memory management operations explicit leads to:

```
LIST function intersect(x:LIST,y:LIST)
var t:LIST; h:INT; z:INT; r:LIST;
begin
   if [x->h t]
   then begin
            z := remove(h,y);
            r := intersect(t,z);
```

```
            detach(z);
            return r;
         end
   else  begin
            attach(y);
            return y
         end
end
```

in which the changes to the function are shown in cursive font. Note that `attach(y)` was introduced for the result value and `detach(z)` was introduced to deal with the temporary value coming from `remove(h,y)` which is stored in `z`. Combining the detach for `z` with the recursive call introduces a new function `intersect'` which can be defined as:

```
LIST function intersect'(x:LIST,y:LIST)
var t:LIST; h:INT; r:LIST;
begin
loop:
   if [x->h t]
   then begin
            x := t;
            r := remove(h,y);
            detach(y);
            y := r;
            goto loop;
         end
   else  return y
end
```

Now the `attach(y)` is no longer needed as its effect is canceled by the `detach(y)` that would have been inserted before `return y`. Using the auxiliary function `intersect'` the original tail-recursive function can now be rewritten to:

```
LIST function intersect(x:LIST,y:LIST)
var t:LIST; h:INT;
begin
   if [x->h t]
   then return intersect'(t,remove(h,y));
   else  begin
            attach(y);
            return y
         end
end
```

5 Critical vs Deferred Reference Counting

In the literature one can find another technique for reducing the overhead of reference counting, known as *Deferred Reference Counting* [Barth77]. Deferred Reference Counting is based on the observation that most references either exist for a very short or for a very long time. Especially references contained in local variables usually have a very short lifetime. They are only created to point to some substructure so that it can be passed to another routine.

The overhead incurred by these temporary references is removed by simply not counting references contained in local variables. The consequence of this approach is that references to a value with zero reference count *can* exist. In order to be able to free and reuse memory, the Deferred Reference Counting method must therefore periodically scan the stack to determine which values with a zero reference count *are* still referenced. To determine the values that are no longer needed, a method must exist to find all values with a zero reference count. For efficiency reasons, a table or list of those values is maintained. Maintaining this table increases the cost of attach and detach operations significantly.

The Critical Reference Counting method reduces the overhead of most temporary references by detecting at compile time that a particular reference can not have any effect on the lifetime of the value referenced. In that case the attach and detach operations can be removed. Usually the attach and detach operations can be expressed in only a few operations, and on most machines are small enough to be inlined. Furthermore, in the Critical Reference Counting system memory is reclaimed as soon as possible, wheras in the Deferred Reference Couting system large amounts of memory may await reclaim. Therefore on both counts (time and space) the Critical Reference Counting system is preferable.

6 Results

As part of an ongoing research project involving a programming language called CDL3 (*Compiler Description Language* see [Kos91]), a classical Reference Counting as well as a Critical Reference Counting garbage collector were implemented. Furthermore, a version with a conservative discontinuous garbage collector was created using a library from Hans-J. Boehm see [BW88,BDS91,Boehm91]. This gasve us the opportunity to compare the effects of using different garbage collectors.

The following measurements (see [Hoedt95]) were done by executing several programs generated with the CDL3 compiler. Using a modified runtime system, activity of the memory management system was reported. The measurements included: amount of memory actually allocated, the number of detaches, the number of detach operations per reclaim operation and total processor time used. Testing several programs, such as the CDL3 compiler, the AGFL compiler, some toy compilers, list manipulation software, finite state simulation software, etc., gave the following results:

1. Programs using the discontinuous memory management system were typically 30-45% slower than the same programs using the Critical Reference Counting memory management system.
2. On average Critical Reference Counting used about the same amount of memory as the Classical Reference Counting method (as expected).
3. The runtimes of the programs increased by 4-12% when using the Critical Reference Counting memory management system instead of no memory management at all (assuming enough physical memory).
4. On the tested programs the Critical Reference Counting method needed only 10-20% of the number of attach and detach operations needed by the Classical Reference Counting method.
5. The ratio of detach operations to the number reclaim operations varied from 3 to 12. Ideally this ratio would be 1, which would mean that the program somehow knew exactly when to reclaim a value. For the Classical Reference Counting method this ratio was between 5 to 9 times as high.

7 Conclusions

In this paper we found that the Critical Reference Counting method needs only 4-12% runtime overhead to perform automatic memory management, which is extremely low. The results indicate that discontinuous memory management systems are not always faster than continuous systems. In fact in our tests the discontinuous memory management system added 30-40% more running time than the Critical Reference Counting system.

The above results show that continuous memory management systems can benefit greatly from information available to the compiler such as life-time information. This paper shows how this information can be used to obtain a highly efficient memory management system. We would like to encourage the use of the Critical Memory Management system in the implementation of other languages. It is a sobering thought that currently a large part of the running time of most complicated programs can be attributed to the memory management system used in the implementation.

The techniques described in this paper do not deal with the problems introduced by using reference semantics. In theory this limitation could be removed by treating values inside compound values in a manner similar to the treatment of global variables. What is needed is a detection algorithm detecting whether a particular subvalue might be modified. An exhaustive global analysis may well turn out to be too costly but enough information might be gathered by using the observation that if the routine being called cannot directly or indirectly get access to a value of the compound type, it can also not modify it. This property can be calculated by examining the module structure.

The problem remains of not being able to reclaim circular datastructures. This problem might be addressed by using a hybrid memory management system where a discontinuous memory management system is periodicaly invoked to find inaccessible circulair datastructures. Such a hybrid system might be appropriate for a Java compiler.

References

[App89] Andrew W. Appel: Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171-183, February 1989.

[App91] Andrew W. Appel: Garbage collection. *Topics in Advanced Language Implementations*, pages 89-100, MIT press, Cambridge, Massachusetts, 1991.

[Axf90] T. H. Axford: Reference Counting of Cyclic Graphs for Functional Programs. *The Computer Journal*, Vol 33, No. 5, pages 466-470, 1990

[Barth77] J. M. Barth: Shifting Garbage Collection Overhead to Compile Time. *Communication of the ACM*, 20(7):513-518, July 1977.

[Boehm91] H. Boehm: Space Efficient Conservative Garbage Collection *ACM SIGPLAN Notices*, 28(6):197:206, June 1993

[BDS91] H. Boehm, A. Demers and S. Shenker: Mostly Parallel Garbage Collection *ACM SIGPLAN Notices*, 26(6):157-164, June 1991

[BW88] H. Boehm and M. Weiser: Garbage Collection in an Uncooperative Environment *Software Practice & Experience*, pp 807-820, September 1988

[Che70] C. J. Cheney: A non-recursive list compactification algorithm. *Communications of the ACM*, 13(11):677-678, November 1970.

[Col60] George E. Collins: A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655-657, December 1960.

[DB76] L. P. Deutsch and D. G. Bobrow: An Efficient, Incremental, Automatic Garbage Collector. *Communication of the ACM*, 19(9):522-526, September 1976

[FY60] Robert R. Fenichel and Jerome C. Yochelson: A LISP garbage collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611-612, November 1969.

[Hoedt95] S. ten Hoedt: *Machine Independent Optimization of CDL3*. Master's thesis no. 358, University of Nijmegen, The Netherlands, August 1995.

[Knu69] Donald E. Knuth: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, chapter 2.3.5, pages 406-422. Addison-Wesley, Reading, Massachusetts, 1969.

[Kop85] Joost van Koppen: *A Reference Mechanism for ELAN*. Master's Thesis no. 8, University of Nijmegen, The Netherlands, September 1985.

[Kos91] C. H. A. Koster: On the Borderline between Grammars and Programs. *Proceedings PLILP'1992*, Passau, 1991.

[LH83] Henry Lieberman and Carl Hewitt: A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419-429, June 1983.

[McC60] John McCarthy: Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184-195, April 1960.

[Wil92] Paul R. Wilson: Uniprocessor Garbage Collection Techniques. *International Workshop on Memory Management*, Springer Verlag Lecture Notes in Computer Science, St. Malo, France, September 1992.